Towards handling 10Pb/s of data through Machine Learning at CERN's Large Hadron Collider

Francesco Derme, francesco.derme@epfl.ch, 394806
Pietro Fumagalli, pietro.fumagalli@epfl.ch, 414991
Saransh Chopra, saransh.chopra@epfl.ch, 407908

In collaboration with the Machine Learning For Experimental Physics team at CERN
Lorenzo Moneta, lorenzo.moneta@cern.ch
Sanjiban Sengupta, sanjiban.sengupta@cern.ch

Abstract—**High Energy Physics (HEP) experiments, such as the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN), produce petabytes of data every second. Physicists are now actively integrating Machine Learning techniques in various parts of the pipeline to collect and analyze this data. Given the massive scale of these experiments, and the upcoming High Luminosity upgrade to LHC (HL-LHC), it's becoming increasingly important to accelerate the inference of ML models beyond the supported capabilities of present day frameworks. The System for Optimized Fast Inference code Emit (SOFIE), is a C++ library developed by CERN for fast ML inference. SOFIE allows parsing a trained ML model into a highly-optimized C++ function, making it possible to run the inference process with minimal overhead and dependencies. CERN's Machine Learning For Experimental Physics team has recently been experimenting with adding heterogeneous computing support to SOFIE using the Alpaka library, allowing it to run inference on any device (including GPUs) while maintaining a single codebase. This paper extends SOFIE's Alpaka backend with four new kernels, and adds related tests and documentation, allowing SOFIE to support inference on GPUs for more ML models. It further benchmarks the newly added operators against PyTorch implementations to showcase an increase in performance and the readiness to be used at scale.**

I. Introduction

With the High Luminosity upgrade to CERN's LHC, starting 2030, physicists are expecting to generate HEP data at the rate of 5 or 10 Petabytes/sec [Gir16] [The17a]. This represents an order-of-magnitude increase compared to current LHC rates, with the detectors effectively becoming 5,000-megapixel cameras taking 40 million pictures every second. The process of collecting and analyzing this data is increasingly based on ML techniques, and this is true throughout a variety of HEP experiments and tasks at CERN, as shown by the following examples.

1) LHC uses physics-informed algorithms to scan the stream of data in real-time, throwing away 98% of it instantly and keeping only interesting physics events [The17b] [Gov+22]. The systems that make this real-time selection are called Triggers. Physicists are now adopting ML techniques at the Level 1 Trigger (L1T) and the High Level Trigger (HLT) in conjunction with traditional algorithms in a bid to find events not predicted by standard physics models.
2) The Compact Muon Solenoid (CMS) experiment uses an advanced neural network model (ParticleNet) for heavy flavor jet tagging and jet mass regression [QG20] [The21] where *jet tagging* is the task of identifying the type of particle that started a jet and *mass regression* is the task of calculating the weight of that parent particle.
3) The ATLAS experiment uses a custom transformer (GN2) for jet tagging [The25] [The23b].
4) Variational autoencoders and transformer-based diffusion models (such as CaloDiT) are used to generate fast calorimeter shower simulations [The23a] [McK+25].
5) To complement fast simulations, physicists are experimenting with clustering and dimensionality reduction models to reduce the size of the data used for training ML models [Fol+24] [RG22].
6) At the offline event analysis side, physicists are adopting gradient-based techniques to optimize parameters and experimenting with treating the whole analysis pipeline as a single, optimizable, ML model [CD19] [SH22].

While training these ML models is well-supported, integrating them at an experiment's runtime requires inference engines that are both extremely fast and hardware-agnostic. SOFIE [MLS+24] [LSM25] is a C++ library that promises to bridge this gap. It generates highly optimized CPU code from trained models (in ONNX format or trained in Keras or PyTorch) for fast and lightweight inference with the BLAS [Bla+02] library as the only external dependency, allowing HEP experiments to integrate complex trained models without the computational overhead of heavy ML frameworks. However, SOFIE currently lacks a unified solution for the diverse landscape of GPU accelerators (NVIDIA, AMD, Intel) used in modern HEP computing centers. Extending SOFIE to support these accelerators traditionally requires writing separate backends for each architecture, but this approach scales badly, leading to code duplication and high maintenance costs. Hence, the Machine Learning For Experimental Physics team has recently started experimenting with Alpaka, a C++ library for parallel kernel acceleration on heterogeneous platforms. Alpaka decouples the algorithmic logic from the hardware layer, enabling a ”write once, run anywhere” approach for parallel kernels. From a technical viewpoint, this project extends SOFIE’s experimental Alpaka backend with more kernels (`Trivial`, `Transpose`, `Concat`, `Where`, and `Topk`), allowing it to run fast inference for a more diverse set of ML models. From a scientific viewpoint, this project has the objective of benchmarking this approach against a state-of-the-art ML framework (PyTorch), with the ultimate goal of evaluating whether it’s fit to handle the challenges and opportunities posed by HL-LHC.

II. Background

SOFIE’s workflow, as illustrated in Figure [1], can be divided into 3 parts

1) Model translation: a trained model from any supported framework (Keras, PyTorch, or the standard ONNX format) is first read by the Parser.
2) Intermediate representation (IR): the Parser converts the complex model structure into a simplified, uniform representation called the RModel.
3) Code generation: the Inference Code Generator takes the RModel and writes the final, highly-optimized, C++ header file with the inference function along with a `.DAT` file containing the weights of the model.

Listing [1] showcases an example of the C++ header file (for our implementation of `Transpose` kernel) generated when an ONNX model composed only of a Transpose layer is passed to SOFIE. The efficacy of this approach has been extensively validated by benchmarking SOFIE-generated CPU code for the ATLAS GN2 model (a Transformer-based architecture for jet tagging) to establish a baseline against ONNX Runtime [LSM25]. Figure [2a] demonstrates that SOFIE consistently minimizes resource usage, requiring significantly less memory than ONNX Runtime (≈130 MB vs 175 MB for 1000 inputs) due to its static memory allocation strategy. Figure [2b] presents how latency scales for the same model. While ONNX Runtime currently exhibits lower latency for large batch sizes on the CPU due to a more mature multi-threading codebase, SOFIE’s performance is heavily dependent on the underlying mathematical backend. The configuration using *Blis* and *VDT* (Vectorized Math Library) provides a measurable speedup over the standard BLAS implementation. It is important to note that HEP trigger systems often prioritize single-event latency optimization (batch size of 1). In this regime, SOFIE’s lightweight design eliminates the overhead of a heavy runtime engine and can already keep up with ONNX Runtime. Building on this high-performance foundation, our project extends the heterogeneous computing support of SOFIE.

III. Methodology

In what follows we give a detailed description of the implemented kernels. Note that *accelerator* refers to the GPU device on which the parallel code generated by Alpaka is executed.

1) `Trivial`: receives as input a tensor with an arbitrary number of dimensions and returns it as output. This requires passing to the accelerator the input tensor and its shape. In the context of this project, this is only useful as a benchmarking baseline.
2) `Transpose`: receives as input a tensor with an arbitrary number of dimensions and swaps two or more dimensions returning a tensor which might be of different size. This requires passing to the accelerator the input tensor as well as desired input and output shapes and the swaps to be performed.
3) `Where`: receives as input 3 tensors with an arbitrary number of dimensions and performs a ternary operator between two of them, using the third as the truth value. This requires passing to the accelerator the input tensors and the input shape.
4) `Concat`: receives as input an arbitrary number of tensors with an arbitrary number of dimensions and concatenates them along a specified axis. The tensors’ dimensions along the concatenation axis must match. This requires passing to the accelerator the input tensors and input shapes, along with the concatenation axis.
5) `Topk`: receives as input a tensor with an arbitrary number of dimensions and finds the biggest `k` elements along a specified axis. If `k` is bigger than the number of items in the chosen axis, the output is padded with a user-defined value. This kernel was templated to accept `k` as well as a `Maxk` as template parameters, so that it can decide to use global memory or GPU registers based on the comparison between the two, allowing for flexibility without sacrificing performance. This requires passing to the accelerator the input tensor and the axis along which the selection shall be made.

IV. Single-kernel tests & benchmarks

Validating the newly implemented parallel kernels requires verifying code correctness and reliability through unit testing, as well as benchmarking performance against established inference engines. The tests were conducted on the three hardware configurations listed in Table [I] in order to assess performance across different backends.

1) The Dell Latitude is a standard x86_64 host using `CpuTbbBlocks` backend.
2) The MacBook Air M3 validates the portability to ARM64 architectures using the same CPU backend.
3) The Lenovo Legion utilizes an NVIDIA GPU with the `GpuCudaRt` backend to demonstrate the performance gains of accelerator offloading (256 threads per block and variable number of blocks).

A. Unit Testing

Single-kernel tests generate the appropriate number of randomly-sized tensors of random values for each kernel (or accept a specified tensor size as command-line-argument), reporting a correctness verdict and the execution time. Correctness is evaluated by comparing the kernel's output to that of a separate, serial implementation that runs on the CPU alone and is guaranteed to be correct because it has been extensively tested. For CPUs, after compiling with the `make` or `make test` commands, tests can be found and run from the `bin` folder. For GPUs, after compiling with the `cmake -S. -Bbuild && cmake --build build` command, tests can be found and run from the `build` folder. We recommend using the `run.py` script described in the next section to handle tests automatically. Testing our implementations several times on every listed architecture with inputs of varying sizes revealed no implementation flaws, consistently producing the expected outputs.

B. Performance Benchmarks

To evaluate the performance and portability of the proposed Alpaka kernels, we developed a Python automation script, `run.py`, which benchmarks the C++ implementations against optimized PyTorch operations. The script manages the compilation pipeline and executes the kernels across a range of input sizes ($N \in \{128, 156, 512, 1024, 2048, 4096, 8192\}$), measuring both kernel execution time and total runtime which includes memory broadcasting overhead from host to device and back. To ensure a fair comparison, the script executes equivalent PyTorch tensor operations on the same accelerator on which Alpaka implementations run. To quantify efficiency, we analyze the *Effective Memory Bandwidth*, which is defined as the total volume of bytes read and written by the accelerator divided by the kernel execution time. This metric isolates memory throughput from computational logic, indicating how effectively the kernel saturates the hardware's memory bus. Additionally, we report the *Speedup*, calculated as the ratio of PyTorch to Alpaka execution times, where a Speedup greater than 1 denotes that the proposed Alpaka kernel outperforms the vendor-optimized baseline. The results for all the implemented kernels are listed in Tables [II][III][IV][V][VI], these were obtained by averaging 10 runs of the `run.py` script. The results are too many to be commented individually, but we highlight general trends. On CPU the Alpaka implementations generally outperform PyTorch on smaller-sized inputs and become slower as the problem size increases, except for *trivial* and *topk* kernels which manage good Speedup even an higher sizes. On GPU on the other hand, Alpaka consistently outperforms PyTorch, except for the *trivial* kernel, demonstrating how powerful this approach can be (keep in mind that CPU and GPU code are compiled from the same Alpaka source code).

V. Conclusion

The integration of Machine Learning into HL-LHC is not merely a performance upgrade, it is a fundamental requirement to sustain the physics potential of the experiment in the High-Luminosity era. With data rates approaching 10 Pb/s, the ability to filter and analyze events with high precision and low latency is paramount. This project has demonstrated that the SOFIE library, augmented by the Alpaka backend, offers a viable path toward hardware-agnostic inference without compromising on performance. By implementing and benchmarking the `Trivial`, `Transpose`, `Concat`, `Where`, and `Topk` kernels, we validated a ”write-once, run-anywhere” approach that successfully decouples the algorithmic logic from the underlying hardware. Our extensive benchmarks across x86_64 CPUs, ARM64 Apple Silicon, and NVIDIA GPUs reveal distinct performance profiles. On GPUs, the Alpaka backend consistently outperforms the optimized PyTorch baseline, achieving massive speedups and confirming the efficiency of the generated CUDA kernels. On CPUs, while PyTorch has the upper hand in high-throughput, large-batch scenarios due to a more-mature multi-threading codebase, our Alpaka implementations demonstrate superior performance in low-latency, small-batch regimes. This is a critical finding, as the Level-1 and High-Level Triggers often operate on single events or small batches where minimizing overhead is more critical than raw throughput. Ultimately, this work confirms that a unified C++ codebase can effectively target heterogeneous architectures. By reducing the maintenance burden of supporting multiple backends and providing competitive inference speeds, the SOFIE-Alpaka integration is well-positioned to support the computationally intensive demands of HL-LHC.

VI. Future work

Most operators, even though written, tested, and benchmarked, have still not been ported to SOFIE. The `Transpose`[1] and `Concat`[2] operators have been ported to SOFIE's repository as two stand-alone branches, but the pull requests are yet to be reviewed by the supervisors and might require additional work to be merged in. Additionally, the kernels must be experimented on with different numbers of threads and blocks on both GPUs and CPUs, to find optimal parameters. Along with these operators, we also experimented with implementing the `ScatterElements`, `Conv`, `BatchNorm`, `Reduce`, and `LayerNorm` operators, but they either require implementing device-agnostic *reduction* operations or device-agnostic BLAS operations, both of which do not have a complete solution as of now. Alpaka does not provide device-agnostic reduction operations, hence, these will have to be added as a stand-alone code in the future. sofieBLAS [MBS+24] attempts to provide users with device-agnostic BLAS operations, but it is still a fairly new library, supporting only limited operations (which are not enough for the operators mentioned above). Finally, once all the above operators are ported to SOFIE, we will be able to generate inference functions for ATLAS GN2 and CMS ParticleNet targeting CUDA, allowing us to benchmark the inference functions and hopefully accelerating their performance by a significant magnitude.

[1] https://github.com/ML4EP/SOFIE/pull/7
[2] https://github.com/ML4EP/SOFIE/pull/8

VII. Ethical risk assessment

Making use of the *Digital Ethics Canvas* [Har+25] to guide our risk assessment, we found *sustainability* to be the critical section. In the context of very large experiments like those carried out at CERN, evaluating energy consumption and environmental impact is non-negotiable. While ML models are a viable (and possibly even necessary) strategy to keep the experiments running under the pressure of the enormous data streams that the HL-LHC will generate, we cannot ignore its potential negative impact. We aim to provide a risk assessment that is grounded in data, thus we ask the reader to excuse the slight technical turn of this section, a high-level summary can be found in [VII]. There are several ways in which the HL-LHC upgrade will impact CERN's energy needs.

1) L1 triggers: the new hardware-level triggers will employ hundreds of high-end FPGA boards. Input bandwidth jumps from 2 Tb/s to 63 Tb/s, processing this volume requires significantly more electricity regardless of the algorithm. It is conservatively estimated that this system will perform 25 billion ML inferences per second. Even at extreme efficiency, this generates a substantial heat load that must be cooled (cooling often doubles the power cost).
2) High-level triggers: this stage is moving toward heterogeneous computing (CPUs + GPUs). Considering that the triggers for the experiment ”Allen” use 500 GPUs to process 5 TB/s and scaling this to HL-LHC levels, we expect the need for a farm of thousands of GPUs.
3) Global computing footprint: the trigger output drives the offline computing needs. It's estimated that computing resources (storage and CPUs) will need to increase by 3–4x by 2030 and 10x by 2041. This implies a massive increase in electricity and hardware manufacturing carbon emissions.
4) Training: while running the trigger takes power, training these massive deep learning models requires repeated passes over exabytes of data. If models are retrained frequently (e.g., every few weeks to adapt to detector aging), the carbon cost of training on large GPU clusters becomes a non-negligible part of the experiment's footprint.

The benefit that ML techniques might bring to HL-LHC is not just better physics, but energy avoidance. By being smarter at the trigger level, the system avoids wasting energy processing and storing *junk* data downstream.

1) Background rejection: At HL-LHC, a *dumb* trigger would be overwhelmed by background noise. To keep the output rate manageable, it would have to raise energy thresholds, missing the Higgs boson or Dark Matter signals entirely. ML algorithms (like Boosted Decision Trees on FPGAs) can distinguish interesting data from noise with 30-50% better background rejection for the same signal efficiency compared to traditional methods. This allows the trigger to lower energy thresholds without losing the physics signal.
2) FPGA vs. GPU efficiency: Recent benchmarks highlight why the FPGA-based ML approach is actually a green choice compared to alternatives: for particle tracking, FPGAs running ML models have shown they can perform inference in microsecond latency with significantly less power per inference than a GPU doing the same task. Furthermore, FPGAs allow for quantization (reducing precision from 32-bit float to fixed-point 6-bit or 8-bit integers). This reduces the resource usage on the chip by factor of 2–4, directly translating to lower power consumption for the same math.
3) Storage reduction: the most energy-expensive action in the experiment is storing data on disk for decades. A smarter ML trigger acts as a high-precision filter. By rejecting background events earlier, the experiment saves the massive energy cost of transmitting, storing, and processing petabytes of uninteresting data.

Is ML a net negative for the environment? Likely no, when viewed holistically. While the power consumption of the trigger racks will undoubtedly rise, the efficiency improves dramatically. Without ML, the LHC would either have to run for many more years to gather the same amount of *good* physics data (costing years of accelerator power), or require an even larger offline computing grid to filter through a noisier dataset. ML in the trigger effectively *front-loads* the energy cost to the most efficient hardware (FPGAs), saving energy in the massive downstream data centers. What is driving the environmental risk is not ML but the HL-LHC upgrade itself but it is not for us in this report to assess whether the physical insight is worth the increased energy consumption. The only helpful thing within our reach was to optimize our kernels to the fullest knowing that at these scales, such optimizations might actually make a noticeable difference in energy consumptions.

VIII. Appendix

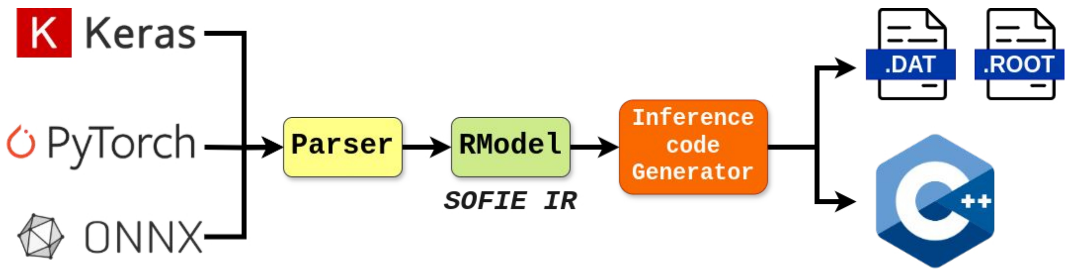
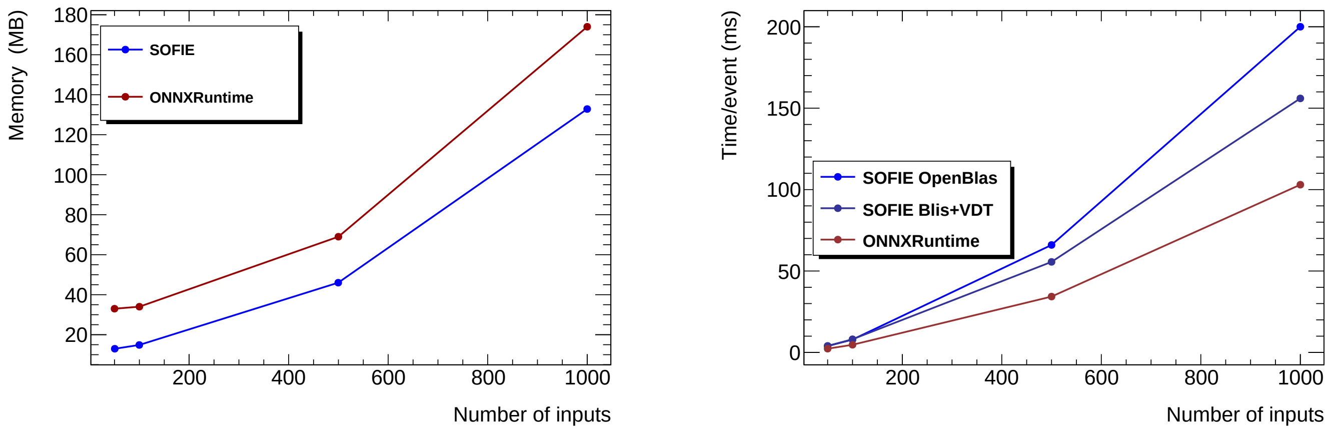


Figure 1. The SOFIE workflow for generating optimized inference code [Lup+25].



(a) Benchmarking memory for ATLAS GN2 [Lup+25].

(b) Benchmarking latency for ATLAS GN2 [Lup+25].

Feature	**Dell Latitude 7400**	**MacBook Air M3**	**Lenovo Legion 5i**
Accelerator Device	Intel Core i7-8665U	Apple M3 (8-core)	NVIDIA GTX 1650TI
Architecture	x86_64 (Whiskey Lake)	ARM64 (Apple Silicon)	CUDA (Turing)
Alpaka Backend	`CpuTbbBlocks`	`CpuTbbBlocks`	`GpuCudaRt`
Compute Units	4 Cores / 8 Threads	4 Perf. + 4 Eff. Cores	1024 CUDA Cores (16 SMs)
Memory Capacity	16 GB DDR4	8 GB Unified Memory	4 GB GDDR6 (VRAM)

Table I
Specifics of testing devices.

Size	Dell Latitude 7400			MacBook Air M3			Lenovo Legion 5i		
	Alpaka GB/s	**Torch GB/s**	**Speedup**	**Alpaka GB/s**	**Torch GB/s**	**Speedup**	**Alpaka GB/s**	**Torch GB/s**	**Speedup**
128	2.06	1.46	1.40	4.49	0.09	48.43	3.5	0.002	1572.72
256	7.70	1.95	3.96	11.35	1.5	7.58	11.88	6.38	1.86
512	16.19	7.57	2.14	8.89	16.48	0.54	27.35	33.03	0.83
1024	11.91	9.05	1.32	10.34	32.06	0.32	44.13	55.5	0.8
2048	13.51	10.51	1.29	5.29	35.79	0.15	67.08	168.96	0.40
4096	13.67	10.62	1.29	4.51	36.17	0.12	69.18	173.02	0.40
8192	13.91	12.34	1.13	2.60	58.78	0.04	61.09	173.45	0.35

Table II
Benchmarking results for **trivial** kernel: memory bandwidth for Alpaka and PyTorch across different architectures and relative speedup.

Size	Dell Latitude 7400			MacBook Air M3			Lenovo Legion 5i		
	Alpaka GB/s	**Torch GB/s**	**Speedup**	**Alpaka GB/s**	**Torch GB/s**	**Speedup**	**Alpaka GB/s**	**Torch GB/s**	**Speedup**
128	1.60	1.64	0.97	2.45	0.16	14.92	4.20	0.03	146.95
256	1.47	4.70	0.31	8.18	5.02	1.63	6.51	89.86	0.07
512	2.24	8.45	0.26	8.94	8.94	1.00	7.72	59.51	0.13
1024	2.54	8.90	0.29	5.12	11.21	0.46	8.14	168.83	0.05
2048	2.86	9.17	0.31	3.43	15.45	0.22	8.49	125.86	0.07
4096	3.01	14.30	0.21	2.86	33.03	0.09	8.64	161.42	0.05
8192	3.12	13.16	0.24	2.15	46.05	0.05	8.49	3.95	2.15

Table III
Benchmarking results for **concat** kernel: memory bandwidth for Alpaka and PyTorch across different architectures and relative speedup.

Size	Dell Latitude 7400			MacBook Air M3			Lenovo Legion 5i		
	Alpaka GB/s	**Torch GB/s**	**Speedup**	**Alpaka GB/s**	**Torch GB/s**	**Speedup**	**Alpaka GB/s**	**Torch GB/s**	**Speedup**
128	0.85	0.21	4.10	0.95	0.01	102.79	0.65	0.00006	9975.93
256	1.05	0.70	1.49	3.62	1.98	1.83	3.41	0.87	3.93
512	1.45	3.21	0.45	14.81	2.17	6.83	8.56	0.14	59.77
1024	2.19	3.84	0.57	25.84	10.87	2.38	20.11	0.11	187.48
2048	3.30	4.05	0.81	31.33	11.11	2.82	29.27	5.88	4.98
4096	4.09	4.10	1.00	38.80	13.60	2.85	36.12	7.86	4.6
8192	4.61	3.67	1.26	39.09	17.41	2.24	37.58	18.15	2.07

Table IV
Benchmarking results for **topk** kernel: memory bandwidth for Alpaka and PyTorch across different architectures and relative speedup.

Size	Dell Latitude 7400			MacBook Air M3			Lenovo Legion 5i		
	Alpaka GB/s	Torch GB/s	Speedup	Alpaka GB/s	Torch GB/s	Speedup	Alpaka GB/s	Torch GB/s	Speedup
128	3.68	1.01	3.65	3.09	0.01	413.43	3.34	0.01	577.86
256	3.14	2.17	1.44	8.39	5.37	1.56	5.02	3.9	1.29
512	2.14	5.78	0.37	9.24	0.91	10.14	26.77	23.58	1.14
1024	2.00	4.85	0.41	9.16	2.76	3.32	52.89	31.51	1.68
2048	0.83	1.80	0.46	9.88	1.06	9.33	66.26	55.37	1.20
4096	0.39	2.29	0.17	5.81	0.92	6.32	59.81	66.77	0.90
8192	0.25	1.85	0.14	4.43	6.38	0.69	62.12	33.71	1.84

Table V

Benchmarking results for **transpose** kernel: memory bandwidth for Alpaka and PyTorch across different architectures and relative speedup.

Size	Dell Latitude 7400			MacBook Air M3			Lenovo Legion 5i		
	Alpaka GB/s	Torch GB/s	Speedup	Alpaka GB/s	Torch GB/s	Speedup	Alpaka GB/s	Torch GB/s	Speedup
128	1.13	0.84	1.35	3.67	0.07	53.88	2.86	0.00006	42241.49
256	1.13	1.87	0.60	7.94	7.33	1.08	17.16	3.41	5.03
512	1.36	4.49	0.30	10.61	5.11	2.08	39.28	126.78	0.31
1024	1.69	6.06	0.28	9.34	2.07	4.52	60.66	48.66	1.25
2048	1.70	6.13	0.28	4.27	2.63	1.62	63.65	143.54	0.44
4096	1.77	6.56	0.27	2.50	2.82	0.89	76.27	170.39	0.45
8192	1.98	6.45	0.31	2.16	62.14	0.03	58.04	172.37	0.34

Table VI

Benchmarking results for **where** kernel: memory bandwidth for Alpaka and PyTorch across different architectures and relative speedup.

Feature	**Current LHC**	**HL-LHC**	**ML impact on energy and environment**
Instantaneous luminosity	2×10^{34} cm^{-2}s^{-1}	7.5×10^{34} cm^{-2}s^{-1}	**Risk:** higher data rate implies a higher base power load
Pileup	$\sim$ 50 collisions	$\sim$ 200 collisions	**Benefit:** ML is essential to efficiently disentangle high pileup
L1 input bandwidth	$\sim$ 2 Tb/s	$\sim$ 63 Tb/s	**Risk:** massive increase in data transport power consumption
L1 output rate	100 kHz	750 kHz	**Risk:** 7.5× more data sent to the HLT farm
Processing tech	Custom electronics + FPGA	High-end FPGA + GPU farms	**Benefit:** FPGAs offer higher energy efficiency per inference compared to CPUs
ML Scale	Minimal	$\sim$ 25 Billion inf/sec	**Risk:** significant thermal load generated by continuous inference

Table VII

Comparison of LHC vs. HL-LHC trigger systems and environmental impact.

```
//Code generated automatically by TMVA for GPU Inference using ALPAKA from Model file
[Transpose.onnx] at [Tue Dec 16 09:27:23 2025]

#ifndef SOFIE_TRANSPOSE
#define SOFIE_TRANSPOSE

#include <vector>
#include <alpaka/alpaka.hpp>
#include <sofieBLAS/sofieBLAS.hpp>
#include "SOFIE/SOFIE_common.hxx"

using Dim1D = alpaka::DimInt<1>;
```

```
namespace SOFIE_Transpose{

//--- ALPAKA Kernels
//------ TRANSPOSE_KERNEL_ALPAKA
struct TransposeKernel {
    template<typename TAcc, typename T>
    ALPAKA_FN_ACC void operator()(TAcc const& acc, T const* input,
    T* output, const std::size_t* input_strides, const std::size_t*
    output_strides, const std::size_t* input_shape,const
    std::size_t* output_shape, const std::size_t* perm, const
    std::size_t ndim) const {
        using DimAcc = alpaka::Dim<TAcc>;
        using IdxAcc = alpaka::Idx<TAcc>;
        constexpr std::size_t D = static_cast<std::size_t>(DimAcc::value);
        alpaka::Vec<DimAcc, IdxAcc> shapeVec{};
        for (std::size_t d = 0; d < D; ++d) shapeVec[d] = output_shape[d];
        auto elements = alpaka::uniformElementsND(acc, shapeVec);
        for (auto const& idx : elements) {
            std::size_t input_idx = 0;
            std::size_t output_idx = 0;
            for (std::size_t d = 0; d < D; ++d) {
                std::size_t out_coord = idx[d];
                std::size_t in_axis = perm[d];
                input_idx += out_coord * input_strides[in_axis];
                output_idx += out_coord * output_strides[d];
            }
            output[output_idx] = input[input_idx];
        }
    }
};


template <typename tagAcc>
struct Session {
    using Idx = std::size_t;
    using Dim = alpaka::DimInt<1>;
    using Acc = alpaka::TagToAcc<tagAcc, Dim, Idx>;
    using DevAcc = alpaka::Dev<Acc>;
    using QueueProperty = alpaka::NonBlocking;
    using QueueAcc = alpaka::Queue<Acc, QueueProperty>;

    alpaka::Platform<Acc> const platform{};
    DevAcc devAcc = alpaka::getDevByIdx(platform, 0);
    alpaka::PlatformCpu platformHost{};
    alpaka::DevCpu hostAcc = alpaka::getDevByIdx(platformHost, 0);
    QueueAcc queue{devAcc};
    Idx threadsPerBlock = 256;

    using Ext1D = alpaka::Vec<Dim, Idx>;
    using Vec = alpaka::Vec<Dim, Idx>;

    //--- declare and allocate the intermediate tensors
    BufF1D deviceBuf_output = alpaka::allocBuf<float, size_t>(devAcc, Ext1D::all(Idx{24}));

    Session(std::string = "") {
        //---- allocate the intermediate dynamic tensors
        alpaka::wait(queue);
    }

   TransposeKernel transposeKernel;

    alpaka::Buf<Acc, float, Dim, Idx> infer(BufF1D const deviceBuf_input){
        //------ TRANSPOSE_GPU_ALPAKA
        alpaka::WorkDivMembers<Dim, Idx>
        workDiv_input(alpaka::Vec<Dim, Idx>::all(24 + 256 - 1) / 256),
        alpaka::Vec<Dim, Idx>::all(256), alpaka::Vec<Dim, Idx>::all(1));
        alpaka::exec<Acc>(queue, workDiv_input, transposeKernel,
        alpaka::getPtrNative(deviceBuf_input),
        alpaka::getPtrNative(deviceBuf_output), {12, 12, 4, 1}, {12, 4, 1, 1}, {2, 1, 3, 4}, {2, 3, 4, 1},
            {0, 2, 3, 1}, 4);

        alpaka::wait(queue);
        return deviceBuf_output;
    }
};   // end of Session
} //SOFIE_Transpose

#endif  // SOFIE_TRANSPOSE
```

Listing 1. Transpose operator generated by SOFIE.

References

[Bla+02] L Susan Blackford et al. “An updated set of basic linear algebra subprograms (BLAS)”. In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[CD19] Pablo de Castro and Tommaso Dorigo. “INFERNO: Inference-Aware Neural Optimisation”. In: *Comput. Phys. Commun.* 244 (2019), pp. 170–179. DOI: 10.1016/j.cpc.2019.06.007. arXiv: 1806.04743 [stat.ML].

[Fol+24] Fritjof Folkeson et al. “Baler - Machine Learning Based Compression of Scientific Data”. In: *EPJ Web Conf.* 295 (2024), p. 09023. DOI: 10.1051/epjconf/202429509023.

[Gir16] Maria Girone. *Big Data Analytics and the LHC*. Keynote presentation at the ACM International Conference on Computing Frontiers (CF’16). 2016. URL: http://helper.ipam.ucla.edu/publications/dmc2017/dmc2017_14378.pdf.

[Gov+22] Ekaterina Govorkova et al. “LHC physics dataset for unsupervised New Physics detection at 40 MHz”. In: *Scientific Data* 9.1 (2022). States: ”This stage rejects more than 98% of the events...”, p. 118. DOI: 10.1038/s41597-022-01187-8.

[Har+25] C. Hardebolle et al. *Digital Ethics Canvas*. 2025. URL: https://www.epfl.ch/education/educational-initiatives/cede/open-and-accessible-education/digital-ethics/a-visual-tool-for-assessing-ethical-risks/the-digital-ethics-canvas-how-to/ (visited on 11/16/2025).

[LSM25] Enrico Lupi, Sanjiban Sengupta, and Lorenzo Moneta. “Benchmark Studies of Machine Learning Inference using SOFIE”. In: *EPJ Web Conf.* 337 (2025), p. 01183. DOI: 10.1051/epjconf/202533701183.

[Lup+25] Enrico Lupi et al. *Enhancements in ML Inference through graph optimizations and heterogeneous architectures*. https://indi.to/hQpPY. 2025.

[MBS+24] Lorenzo Moneta, Andrea Bocci, Sanjiban Sengupta, et al. *sofieBLAS: an abstract C++ (header-only) interface for BLAS operations targeting heterogeneous architectures*. https://github.com/ML4EP/sofieBLAS. 2024.

[McK+25] Peter McKeown et al. *A Generalisable Generative Model for Multi-Detector Calorimeter Simulation (CaloDiT)*. 2025. arXiv: 2509.07700 [hep-ex].

[MLS+24] Lorenzo Moneta, Enrico Lupi, Sanjiban Sengupta, et al. *SOFIE: System for Optimized Fast Inference code Emit*. https://github.com/ML4EP/SOFIE. 2024.

[QG20] Huilin Qu and Loukas Gouskos. “ParticleNet: Jet Tagging via Particle Clouds”. In: *Phys. Rev. D* 101.5 (2020), p. 056019. DOI: 10.1103/PhysRevD.101.056019. arXiv: 1902.08570 [hep-ph].

[RG22] Harry Röhrich and Jochen Gemmler. “The Federation – A novel machine learning technique applied to data from the Higgs Boson Machine Learning Challenge”. In: *Proceedings of the 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2022)*. 2022. URL: https://indico.cern.ch/event/1106990/papers/4998136/.

[SH22] Nathan Simpson and Lukas Heinrich. “NEOS: Neural End-to-End Optimized Statistics”. In: *Mach. Learn.: Sci. Technol.* 3.2 (2022), p. 025008. DOI: 10.1088/2632-2153/ac64ba.

[The17a] The ATLAS Collaboration. *Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System*. Technical Design Report CERN-LHCC-2017-020; ATLAS-TDR-029. CERN, 2017. URL: https://cds.cern.ch/record/2285584.

[The17b] The CMS Collaboration. “The CMS trigger system”. In: *Journal of Instrumentation* 12.01 (2017). Describes the hardware-based physics algorithms (Level-1), P01020. DOI: 10.1088/1748-0221/12/01/P01020.

[The21] The CMS Collaboration. *ParticleNet jet mass regression*. CMS Detector Performance Note CMS-DP-2021-017. CERN, 2021. URL: https://cds.cern.ch/record/2777008.

[The23a] The ATLAS Collaboration. “Deep generative models for fast photon shower simulation in ATLAS”. In: *Comput. Softw. Big Sci.* 7.1 (2023), p. 10. DOI: 10.1007/s41781-023-00106-9.

[The23b] The ATLAS Collaboration. *Transformer Neural Networks for Identifying Boosted Higgs Bosons decaying into $b\bar{b}$ and $c\bar{c}$ in ATLAS*. ATLAS Public Note ATL-PHYS-PUB-2023-021. CERN, 2023. URL: https://cds.cern.ch/record/2866601.

[The25] The ATLAS Collaboration. “Transforming jet flavour tagging at ATLAS”. In: *Nature Communications* in press (2025). arXiv: 2505.19689 [hep-ex].